Using Squeak for Teaching User Interface Software

Mark Guzdial
College of Computing
Georgia Institute of Technology

Abstract

Squeak is a new programming language based on an old one that skipped some 15 years of development. Squeak is highly cross-platform, running on Windows, Macintosh, Linux, BeOS, and Windows CE devices (among others) bit-identically. It has been updated with modern features, such as web browsing and serving, 3-D graphics engine, and powerful sound synthesis. Squeak is an excellent pedagogical platform because it doesn't presume a windowing operating system. Instead, Squeak implements all of the windowing and other user interface software itself, providing both a rich set of examples and a bare substrate on which one can explore and build user interfaces from scratch. We have used Squeak both to enhance the infrastructure for our course, and to change how we teach user interfaces. We present a pilot study suggesting benefits of the new approach.

1 Squeak's Beginnings

Smalltalk-80 was developed by the Learning Research Group (LRG) at Xerox PARC, and is widely viewed as the first system to implement the desktop user interface. LRG was headed by Alan Kay and included members Dan Ingalls, Ted Kaehler, Adele Goldberg, and others. Smalltalk-80 included icons, a mouse pointer, pop-up menus, and a complete windowing system implemented in one of the first object-oriented programming languages.

In 1981, Xerox PARC sent tapes of *Smalltalk-80* to several computer companies, to disseminate the technology and to test the portability of Smalltalk-80. Smalltalk was implemented as a bytecode compiler for a virtual machine. One of the companies that received Smalltalk-80 was Apple Computer, who did successfully port Smalltalk-80 into Apple Smalltalk.

In 1995, Alan Kay, Dan Ingalls, and Ted Kaehler were together again at Apple, and they decide that they need for their own research a kind of Smalltalk that didn't currently exist [3]. What they wanted was a completely cross-platform, open source Smalltalk (in order to take advantage of the distributed power of programmers all over the Internet) that was rich in media supports. They didn't have to start from scratch, because they still had that Apple Smalltalk from the original Xerox Smalltalk-80.

To gain the cross-platform flexibility they wanted, they re-wrote the virtual machine in Smalltalk, then wrote a Smalltalk-to-C translator in order to generate ANSI standard C for compiling a fast VM. When they released this new Smalltalk, *Squeak*, to the Internet in October 1995 they only had a Macintosh port, but within a month, it had been ported to UNIX and Windows. Today, Squeak[1] runs on virtually any platform, and thousands of open source developers extend it, along with "Squeak Central" (now moved to Disney Imagineering R&D from Apple).

We at Georgia Tech began teaching with Squeak in 1998 in our Sophomore required course on object-oriented analysis and design emphasizing user interface design and implementation. We had been using ParcPlace VisualWorks, because we preferred the pure objects of Smalltalk to C++. But VisualWorks didn't run on all the platforms that our students were using (e.g., Linux) and was quite expensive, we were looking for an alternative.

What we found in Squeak, however, was the opportunity to teach user interfaces in a new way. Because Squeak does not presume a windowing user interface, all the windowing software is written in Squeak. That

[1] `http://www.squeak.org`

Figure 1: Drawing a line across window boundaries in Squeak

means that it's possible to write over the windows (Figure 1) and even to construct one's own windows–or to program user-interfaces *without* windows. It's this latter capability which we have used to change how we teach user interface software.

2 Squeak in Objects and Design

The course in which we use Squeak is an introduction to *Objects and Design*. Students in this class have already had[2]:

- A one semester *Introduction to Computing*
- A one semester *Introduction to Object-Oriented Programming* in Java.
- A one semester course on *Languages and Translation* in C using tools like LEX and YACC to explore the issues of language implementation, from models of the bare processor up through tokenizing and parsing.

The goal of the *Objects and Design* course is to explore higher-level issues of design. The class is large: Typically 100 or more students a semester. We introduce an object-oriented design process, which starts from analysis (with CRC Cards) and leads through design (using UML class diagrams). The course also serves as an introduction to the issues of user interface implementation and design. Modern user interfaces grew up with object-oriented programming (starting in Smalltalk), and using user interfaces to explore object-oriented concepts (e.g., aggregation, composition, inheritance, delegation) is natural and offers concrete examples. For example, how windows interact with their component objects allows for exploration of how messages get passed between peer objects, with the opportunity of visual experiments.

The course centers around a semester-long, team project. Because we're using Squeak with its rich multimedia support, the team project often involves interesting manipulation of media. In one semester, students had to build personalized newspapers where stories were

[2]This description focuses on the current state of the class under semesters. Previous to Fall 1999, Georgia Tech was under the quarter system, but a similar class was offered with similar pre-requisite classes

Figure 2: A CoWeb – normal view on left, and edit view on right

drawn from web-sites and laid out (multi-column with graphics). To encourage flexible designs, student teams are asked to serve their applications through more than one kind of interface. In the newspaper example, students had to be able to serve the newspaper via a Web interface (served from Squeak's internal webserver) and via PostScript (generated using Squeak's PostScript Canvas).

2.1 Using Squeak for Infrastructure and Case Study

My research is on computer-supported collaborative learning (CSCL), with an emphasis on communicating through multimedia, where Squeak is a natural platform. By using Squeak in the course as well, we not only built upon the developing experience of myself and my graduate students, but we also set up for an interesting use of Squeak as infrastructure. Students in *Objects and Design* use our tools as part of the normal practice of the class, and then we use the tools as a case study to critique the tools' interface and object design.

One of the tools that we use in *Objects and Design* is the *CoWeb* (Collaborative Website), also known as *Swiki* since it's a Squeak interpretation of Ward Cunningham's WikiWiki Web[3]. The CoWeb is perhaps the simplest possible collaboration tool: Every page is editable by anyone (via an edit link on the page), and anyone can easily create new pages and links between pages (Figure 2). By typing `*A New Page*` on any page, a new page is created and linked in with the name `A New Page`. Surprisingly, such a structure does *not* lead to anarchy. Instead, it is now in use by some 120 groups at Georgia Tech, across ten servers, and is being adopted by other schools around the world[4].

We use the CoWeb in several ways in *Objects and Design*[5].

- Students are asked to create pages for themselves with

[3]`http://w2.com/cgibin/Wiki`

[4]We release CoWeb as an open-source project. For more information, see `http://pbl.cc.gatech.edu/myswiki`

[5]`http://coweb.cc.gatech.edu/cs2340`

their own name (e.g., by typing `*Mark Guzdial*` on the *Who's Who* page). From then on, they can "sign" their postings with their name, and that name links to their page where they can introduce themselves.

- Each assignment has its own Q&A page. Such a persistent structure as conversation on a Web page has been shown to lead to more extended discussion [2].
- We also have a number of small activities, like the *Surprises* page where students are asked in their third or fourth week of class to leave notes to the next class about what they wish they had realized in the first week of class.
- The most popular activities in the CoWeb are the exam reviews. For each exam, a set of example problems is posted with a question/comment page linked to each problem. Students are welcome to ask for help on the problem, post solutions, or critique each other solutions. The pages are monitored by teachers and teaching-assistants, but the "correct" answers are never posted by them. Wrong answers are pointed out, but correct answers are met often with silence or "Yes, that would work, but there are other (perhaps better) solutions." This encourages students to plug away at a problem (not just memorize the first answer), yet provides useful feedback. In interviews and surveys (as well as the raw measure of participation rates), students identify this as one of the most useful activities in the CoWeb. In a sense, it uses CSCL to create a class-wide study group.

Since the CoWeb is in Squeak (built on a webserver also in Squeak), we are able to use it as a case study. Typically, we disassemble the CoWeb before students have to build their own web interfaces. We critique the object design of the CoWeb and its user interface. For example, the original CoWeb was not well-designed for the variety of different looks-and-feels that users want in their CoWebs, so it serves as a point of discussion for how to create flexible object structures. Also, the interface of the CoWeb is based around HTML, which has not been welcoming to Mathematics classes whose central medium (equations) are difficult to express in HTML. There is a definite synergy about critiquing a tool which the students themselves use, which they have complete access to, and which they can take apart and reuse in their own class projects.

2.2 Using Squeak for Teaching UI Software

Since we have moved to Squeak, we have been able to change how we teach user interfaces, to good effect. One of the challenges of teaching user interface software is helping students to understand the *Model-View-Controller* (MVC) paradigm. Basically, MVC describes a mechanism for flexibly connecting user interfaces to underlying object structures.

- Models are the application objects, drawn from an analysis of the problem domain.
- Views are the user interface objects, including buttons, text areas, and the like.
- Controllers are objects that mediate user interface events (like mouse moves and keystrokes) and transfer them to the appropriate objects.

MVC has always been difficult for students to understand. John Carroll and his team identified this problem in their work teaching Smalltalk in the 80's [1]. The problem is the complexity of a loosely-coupled system. Students want models to directly control views (or vice-versa), but the MVC paradigm creates layers of indirection which allow models and views to be modified separately.

We identified this problem early on, when we first started teaching this course in 1993 under the quarter system. We began tracking performance, by using similar problems on midterm examinations and comparing the results across terms. For example, in the Winter 98 midterm exam, students were asked to design part of the objects for a phone system, and then asked in a separate problem:

> **Displaying the Incoming Phone Number**. The central office now gives you a new feature: You can ask a network connection for its phoneNumber. Now, its possible to have a View on the phone that displays the phone number for the incoming call.
>
> How would you change your Phone class (or others) to take advantage of this feature, assuming an MVC approach? You can assume that the Phone is subclassed off Model. Be sure to tell me (a) how the PhoneNumber View finds out about an incoming call (e.g., once the call arrives at the Phone instance, how does the View know that it needs the phone number?) and (b) how the View finds the number to display (e.g., what does the View do to get the phone number?).

The correct answer to this problem was to identify that the model broadcasts a *change* to all of its views, and the views query the model for the updated value (in this case, a phone number). Partial credit was provided for any part of this answer, e.g., that models broadcast to the views, but without stating that a "change" is broadcast (an important aspect of MVC, since the model should not presume to send the actual data to the views and thus create a constraint on what the views

Table 1: Results of Traditional Instruction on MVC

Term (Number of Students)	Average	StdDev
Spring97 (86)	0.44	0.36
Summer97 (48)	0.54	0.40
Winter98 (107)	0.54	0.34

might want given the particular change in the model). This exact same problem was used in the Spring 1997 midterm exam. An alternative form of the problem was used in the Summer 1997 and Spring 1998 midterm exams. After designing the objects for an alarm system, students were asked:

> **Alarm Status System**. You are now designing the alarm status monitoring systemthat is, an interface to watch over the alarms. You decide to use a Model-View-Controller structure for your system. You decide that you will have an Alarm class which will be your model, and an AlarmView for displaying the status. Someone else is building your controller – you dont worry about that.
>
> A. How does the Alarm object alert the AlarmView that it must update?
>
> B. How does the AlarmView get the alarm status to be displayed?

During the Spring 1997, Summer 1997, and Winter 1998, we used Coad and Nicola's *Object-oriented programming* (1993, Prentice-Hall) with VisualWorks. We taught MVC with examples and explanation. The average and standard deviation for the MVC Midterm Problem in each of these terms appears in Table 1.

The Spring 1998 term, however, was the first one in which we tried teaching user interfaces in a new way. Rather than simply describe MVC with examples, we "built" MVC in class with live walkthroughs of code and live demonstrations of the results. We led students through three iterations of a user interface for an application already designed in class. In all three iterations, we used no existing UI classes. Instead, we drew graphics (using the Logo-turtle-like `Pen` object in Squeak) directly on the `Display` object, and we read user interface events by polling the `Sensor` object.

- In the first iteration, we did the most obvious thing: We hacked the application objects to create a user interface. We introduced the concept of an *event loop* to poll for events and dispatch them appropriately. But the event loop and all display objects were smack in the middle of our models, which defeats good object-oriented principles of appropriate responsibility.
- In the second iteration, we split off *window* and *button* objects. Now, the event loop sat inside the window which dispatched the events into the buttons. The buttons sent messages to the model objects. But the updating of text feedback from the model (as described in the midterm problems) was still occurring within the models themselves.
- In the third iteration, we introduced the *changed-update* broadcast mechanism in Squeak which relates models and views by dependencies, not directly. Still without using any existing UI classes, we added a *text* view to our window and button classes, creating a tiny but relatively modern mini-UI system.

Table 2: Results of New Approach Midterm Problem on MVC

Term (Number of Students)	Average	StdDev
Spring98 (103)	0.86	0.27

The results in the Spring 1998 midterm were markedly different on the Alarm version of the problem, as seen in Table 2. We were able to use the exact same graders and exact same grading scheme between the Winter and Spring 1998 terms. We performed a t-test comparing the two terms and found the difference to be significant $p < 4.4E-13$.

This is not a rigorous experiment by any means. We are not controlling for all other variables. The Spring 1998 students may just have been better than the Winter 1998 students. Our intuition is that that's not true. We did compare other problems of a similar nature between the two terms, and found that on other problems, the average favored the Winter 1998 students. But it's still possible that the Spring 1998 students were more amenable to learning MVC.

Nonetheless, the findings are promising and suggest that continuing with this approach makes sense. It's worth noting that replicating the approach of building user interface toolkits from scratch is fairly hard to do in most other languages and operating systems. Most operating systems do not allow you to write directly to the screen without a lot of low-level hacking. While it's certainly possible to get a window from the operating system and write new windowing software within that window (which is essentially what Squeak does), the reality that all that code already exists in Squeak makes it appealing. Further, Squeak works on virtually all modern platforms, which means that individual windowing system differences don't enter into the problem as they might.

References

[1] Carroll, J. M., Singer, J., Bellamy, R., and Alpert, S. A view matcher for learning smalltalk. In *Pro-*

ceedings of CHI'90: Human Factors in Computing Systems, J. Chew and J. Whiteside, Eds., vol. Seattle, April 1-5. ACM Press, New York, 1990, pp. 431–437.

[2] Guzdial, M., and Turns, J. Effective discussion through a computer-mediated anchored forum. *Journal of the Learning Sciences*, To appear (2000).

[3] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*. ACM, Atlanta, GA, 1997, pp. 318–326.